C programming

Creator

Dennis Ritchie

Learning verse

-Happy learning

Introduction

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone laboratories to develop the Unix operating system. C is the most widely used computer language.

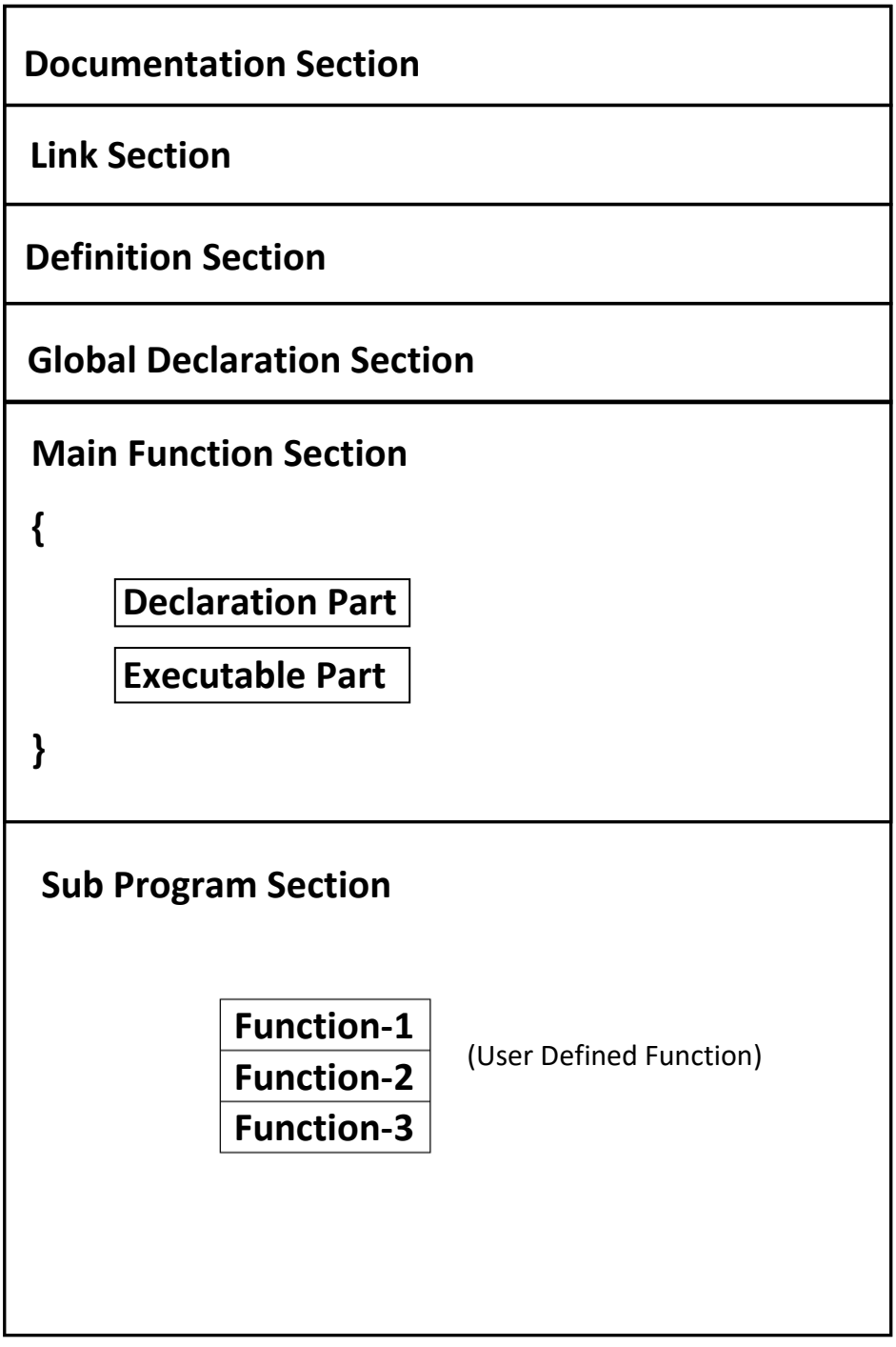
Facts about C

1. C was invented to write an operating system called Unix.
2. C is a successor of B language which was introduced around the early 1970s.
3.This language was formalized in 1988 by the American National Standard Institute (ANSI).
4. The UNIX OS was totally written in C.
5. Today C is the most widely used popular System Programming Language.

History of C language

1960	ALGOL	International group
1967	BCPL	Martin Richards
1970	B	Ken Thomson
1972	C	Dennis Ritchie
1978	K & RC	Kernighan & Ritchie
1989	ANSI C	ANSI Committee
1999	C99	Standard Committee
2011	C11	Standard Committee

General Structure of C language



- The documentation section is used for displaying any information about the program like the purpose of the program, date, time and this section should be enclosed within

comment lines. The statements in the section are ignored by the compiler.

- The link section consists of header files.
- The definition section consists of macro definitions, defining constants etc.
- Anything declared in the global declaration section is accessible throughout the program and it can be accessible to all the functions in the program.
- Main() function is mandatory for any program and it includes two parts, the declaration part and the executable part.
- The last section, sub-program section is optional and used when we require including user defined functions in the program.

Syntax

- **Semicolons**: In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
  Ex: printf(“Hello world”);
- **Comments**: Comments are like helping text in the C program and they are ignored by the compiler. They start with /* and terminate with the characters */
- **Identifiers**: A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).
  Whitespace: Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. It separates one part of a statement from another and enables the compiler to identify where one element in a statement, and the next element begins.

Comments

- Comments can be used to explain code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

- Comments can be **single-lined** or **multi-lined**.

Single – line comments:

Single-line comments start with two forward slashes (//). Any text between // and the end of the line is ignored by the compiler (will not be executed).

Ex: // This is a comment
printf("Hello World!");

Multi – line comments:

Multi-line comments start with /* and ends with */. Any text between /* and */ will be ignored by the compiler.

Ex: /* The code below will print the words Hello World!
to the screen, and it is amazing */
printf("Hello World!");

Variables

Variables are containers for storing data values. To indicate the container, each variable should be given a unique name (Identifier). Variable names are just the symbolic representation of a memory location.

For example: int score=95;
Here, score is variable of int type. Here, the variable is assigned an integer value 95.

Rules for naming a Variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Declaring variables
To create a variable, specify the **type** and assign it a **value**.

Syntax: *type variableName* = *value*;

Where *type* is one of C data types (such as int), and *variableName* is the name of the variable (such as x, y). The **equal sign** is used to assign a value to the variable.

Ex: int myNum = 15;

Here int is the data type, myNum is the name of the variable. The equal sign is used to assign 15 to the variable name myNum.

Data types

Types	Data types
Basic data type	Int, char, float, double
Derived data type	Array, pointer
User-defined data type	Structure, union, enum

Basic data type

The basic data types are integer-based and floating-point based.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

Derived data type

Derived data types are derived from the primitive or fundamental data types. There are mainly 3 types of derived data types in C.

Array:

An array is a group of similar kinds of finite entities of the same type. These entities or elements can be referred to by their indices respectively. The indexing starts from 0 to (array_size-1) conventionally. An array can be one-dimensional, two-dimensional, or multidimensional.

Syntax: data_type arr_name[size];
Description of the syntax
Data_type: The data type specifies the type of elements to be stored in the array. It can be int, float, double, and char.
array_name: This is the name of the array. To specify the name of an array, you must follow the same rules which are applicable while declaring a usual variable in C.
size: The size specifies the number of elements held by the array. If the size is n then the number of array elements will be n-1.

For Ex:

```
#include <stdio.h>
int main() {
  int myNumbers[] = {25, 50, 75, 100};
  printf("%d", myNumbers[0]);

  return 0;
}
```

OUTPUT
25

Declaring arrays

- To declare an array in C, a programmer specifies the type of the elements and the number of elements required in an array

Syntax: Type arrayName[arraySize];

- This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type.

Access the elements of an array

- To access an array element, refer to its **index number**. Array indexes start with **0**.[0] is the first element, [1] is the second element, etc.

```
Ex: int myNumbers[] = {25, 50, 75, 100};
   printf("%d", myNumbers[0]);
```

Change an element in an array

To change the value of a specific element, refer to the index number.

```
Ex: int myNumbers[] = {25, 50, 75, 100};
      myNumbers[0] = 33;
      printf("%d", myNumbers[0]);
```

OUTPUT

33

Pointer:

A pointer can be defined as a variable that stores the address of other variables. This address signifies where that variable is located in the memory. If a is storing the address of b, then a is pointing to b. The data type of a pointer must be the same as the variable whose address it is storing.

Syntax: type *pointer_name;

Description of the Syntax

type: This is the data type that specifies the type of value to which the pointer is pointing.

pointer_name: This is the name of the pointer. To specify the name of a pointer, you must follow the same rules which are applicable while declaring a usual variable in C. Apart from these rules, a pointer must always be preceded by an asterisk (*).

For Ex:

```
#include <stdio.h>
int main() {
  int myAge = 43;
  printf("%d\n", myAge);
  printf("%p\n", &myAge);
  return 0;
}
```

OUTPUT

43

0x7ffe5367e044

Dereference

In the example above, we used the pointer variable to get the memory address of a variable (used together with the & **reference** operator).

However, you can also get the value of the variable the pointer points to, by using the * operator (the **dereference** operator):

```
int *ptr = &myAge
printf("%p\n", ptr);//Reference
printf("%d\n", *ptr);//Dererence
```

Changing value pointed by the pointers

```
Ex: int* pc, c;
    c = 5;
    pc = &c;
    c = 1;
    printf ("%d\n", c);
    printf("%d", *pc);
OUTPUT
1
1
```

Difference between Array and Pointer

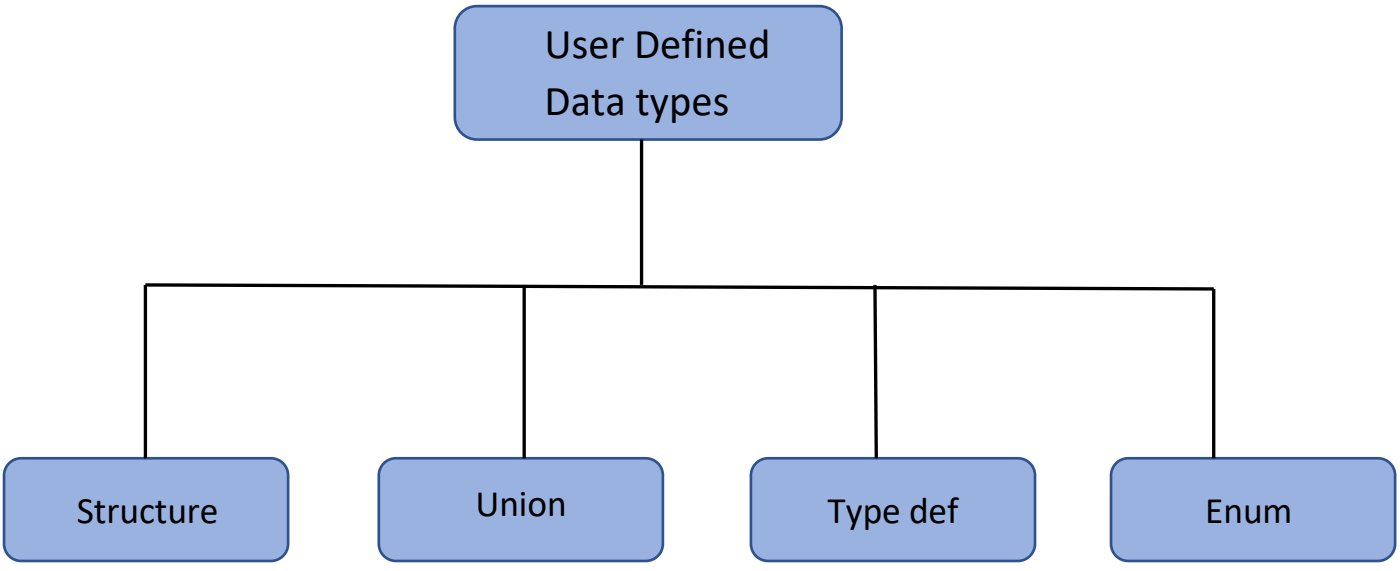
Array	Pointer
A data structure consisting of a collection of elements each identified by the array index	A programming language object that stores the memory address of another value located in company memory
Refers to a set of data elements	A variable that points some other memory location
Array syntax – data type arrayName[data type];	Pointer syntax – data type* variable_name;
Used to allocate fixed size memeory (static memory)	Used for dynamic memory allocation

User defined data type:

"The user-defined data types in C" is a collection of data with values that have fixed meanings. A data type in C is a predefined value that helps declare, access, and store data. The user-defined data types in C can be derived from C's primary or built-in data types.

Why do we Need User-Defined Data Types in C?

- User-defined data types in C are highly customizable, depending upon the use case of the programmer.
- User-defined data types in C increase the readability of the program as data types can be given meaningful names.
- User-defined data types increase the extensibility of the program to introduce new data types and manage data in the application code.
- If we do not use user defined data types in C, then it is very difficult to maintain multiple variable that represent information belonging to the same entity in the program logic.



Structure

A structure is a user-defined data type in C that allows to combine members of different types under a single name (or the struct type). The reason why it is called a user-defined data type is that the variables of different types are clubbed together under a single structure, which can be defined according to the user's choice.

Syntax:

```
struct structure_name
{
   data_type var1;
   data_type var2;
};
```

Description of Syntax

struct: The definition of a structure includes the keyword struct followed by its name. All the items inside it are called its members and after being declared inside a structure.

data_type: Each variable can have a different data type. Variables of any data type can be declared inside a structure.

The definition of a structure ends with a semicolon at the end.

Create a Structure

You can create a structure by using the struct keyword and declare each of its members inside curly braces:

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Access Structure Members

To access members of a structure, use the dot syntax (.):

```
#include <stdio.h>
// Create a structure called myStructure
struct myStructure {
    int myNum;
    char myLetter;
};
int main() {
    // Create a structure variable of myStructure called s1
    struct myStructure s1;
    // Assign values to members of s1
    s1.myNum = 13;
    s1.myLetter = 'B';
    // Print values
    printf("My number: %d\n", s1.myNum);
    printf("My letter: %c\n", s1.myLetter);
    return 0;
}
```

OUTPUT

```
My number: 13
My letter: B
```

Union:

A union is also a user-defined data type. It also holds members of different data types under a single name. A union sounds similar to a structure and they are similar in conceptual terms. But there are some major differences between the two. While a structure allocates sufficient memory for all its members, a union only allocates memory equal to its largest member.

Syntax

```
union structure_name
{
   data_type var1;
   data_type var2;
};
```

Description of Syntax

union: The union keyword is written at the beginning of the definition of a union in C. After it, the name of the union is specified.
data_type: It is the data type of the member variable of the union. Members of different types can be defined inside a union.

Ex:

```
union Book{
        char name[100];
        int pages;
};
int main(){
        union Book myBook;
        myBook.pages=10;
        printf("No of pages in Book=%d\n",myBook.pages);
}
```

OUTPUT

10

Enum:

Enumeration or simply enum is one of the user-defined data types in C which provides a special type of flexibility of defining variables. An enum consists of a set of integer constants that can be replaced by user-defined names.

Description of Syntax

enum: The keyword enum is written at the beginning of the definition.

flag: This is the default name of the enumeration set. It can be replaced by another name or can be used as it is.

const_name: It is the integral identifier inside the enum set. The default values of this set are- {0, 1, 2, ….}.

Ex:

```
enum month {
   // Create enum with default values assigned by compiler.
   JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
   // JAN = 0, FEB = 1, APR = 2 and so on.
};
enum monthDays {
   // Values are assigned to constants.
   JANUARY = 31, FEBRUARY = 28, MARCH = 31
};
enum colors {
   // Starting value is assigned.
   RED = 1, GREEN, BLUE
   // RED = 1, GREEN = 2, BLUE = 3
};
int main() {
   printf("January Month Number: %d\n", JAN);
   printf("Number of days in January: %d\n", JANUARY);
   printf("Favorite  Color: %d\n", BLUE);
}
```

Type Def

Type definitions allow users to define an identifier that would represent a data type using existing or predefined data types. It is used to create an alias or an identifier for an existing data type by assigning a meaningful name to the data type. The identifier is defined using the keyword typedef. The main advantage of using typedef is that it allows the user to create meaningful data type names. This increases the readability of the program.

Declaration

typedef type identifier
OR
typedef existing_datatype_name new userdefined_datatype_name
Ex:

```
// Assign the name "Book" to the structure defined below using typedef.
typedef struct {
   char name[100];
   char author[100];
} Book;
// Assign an identifier "number" to the existing data type "int"
typedef int number;
// Create an alias "decimal" for the data type "float".
typedef float decimal;
// Now we can use above defined names as data types to create variables.
int main() {
   // Create variables using the identifiers.
   number a = 10;
   decimal b = a * 1.5;
   Book myBook;
}
```

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators:

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description
+	Addition	Adds together two values
-	Subtraction	Subtracts one value from another
*	Multiplication	Multiplies two values
/	Division	Divides one value by another
%	Modulus	Returns the division remainder
++	Increment	Increases the value of a variable by 1
--	Decrement	Decreases the value of a variable by 1

Relational operators:

Operator	Description
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.

Logical operators:

Operator	Name	Description
&&	Logical and	Returns true if both statements are true
||	Logical or	Returns true if one of the statements is true
!	Logical not	Reverse the result, returns false if the result is true

Bitwise operators:

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, | and ^ is as follows

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Assignment operators:

Assignment operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Conditional Statements

If statement

- The if statement evaluates the expression inside the parenthesis ().
- If the test expression is evaluated to true, statements inside the body of if are executed. If the test expression is evaluated to false, statements inside the body of if are not executed.

Syntax:

```
if(expression) {

        //code

}
```

If – else statement

The if statement may have an optional else block.

Syntax:

```
If (test expression) {
   //code runs if the test expression is true
}
else {
  //code runs if the test expression is false
}
```

If…else ladder

The if…else statement executes two different codes depending upon whether the test expression is true or false. It allows us to check between multiple test expressions and execute different statements.

Syntax:

```
If (test exp1) {
        //statement
}
else if (test exp2) {
        //statement
}
```

```
else if (test exp3) {
        //statement
}
.
.
else {
        //statement
}
```

Switch statement

Instead of writing many if…else statement, we can use switch statement. The switch statement selects one of many code blocks to be executed.

Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. A loop statement allows us to execute a statement or group of statements multiple times.

- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.

While loop

While loop, repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax:

```
while (condition) {
  // code block to be executed
}
```

Ex:

```
#include <stdio.h>
int main() {
  int i = 0;
  while (i < 5) {
    printf("%d  ", i);
    i++;
  }
  return 0;
}
```

OUTPUT

```
0 1 2 3 4
```

For loop

For loop, execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.
Statement 2 defines the condition for executing the code block.
Statement 3 is executed (every time) after the code block has been executed.
Ex:

```
#include <stdio.h>
int main() {
  int i;
 for (i = 0; i < 5; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

Do-while loop

Do-while is like a 'while' statement, except that it tests the condition at the end of the loop body.

Syntax:

```
do {
  // code block to be executed
}
while (condition);
```